5th May 2015

Intelligent Agent for Open Face Chinese Poker Web-Application

Submitted May 2015 in partial fulfilment of the conditions of the award of the degree BSc (Honours) Computer Science

Alastair Kerr

axk02u

School of Computer Science and Information Technology
University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature ________________

Date ____/____/____

Abstract

Artificial Intelligence (AI) is a crucial and high-interest area of research in the field of Computer Science, which has gained increased traction in recent decades due largely to the requirements for increasingly sophisticated AI in games, and the implications that development of *algorithms* and *heuristics* in this area can have on other fields of study such as *Automated Planning and Scheduling.* Poker games provide a challenge to *Intelligent Agents* because of many factors including *combinatorial explosions* of *game trees*, elements of *hidden information* such as the cards other players hold, as well as stochastic elements due to not knowing which cards are yet to be dealt. This differentiates Poker games from more traditional games such as Checkers, which is a deterministic *perfect information* game meaning that each player has the same complete knowledge of the game state at any stage and that there is a finite set of moves for each player. Therefore at each stage there is an *optimal move* leading to a winning strategy, and in this manner these games can effectively be *solved*, and so when playing against a competent Agent for such a game it is impossible to win, only to draw. Poker games are different because of the aforementioned stochastic elements, imperfect information and necessarily complex search trees, necessitating the use of more sophisticated algorithms in order for an Agent to perform competently versus intelligent opponents. While there has been lots of research into Intelligent Agents for traditional board games and variants of Poker such as *Texas Hold'Em*, lesser known variants such as the relatively new *Open Face Chinese Poker* have not been explored to the same degree. Agents for Open Face Chinese Poker often suffer from poor performance due to reliance on simple algorithms and methods such as *Rule-Based Systems*, which can lead to predictable or sub-optimal play that is additionally largely domain specific. This dissertation considers the merits and limitations of various AI techniques, and implements an Intelligent Agent for a bespoke Open Face Chinese Poker Web-Application, with discussions of the range of technologies used and methodologies employed in creating a functional final product. It is advisable for readers unfamiliar with Poker, Computer Science or any of the italicised terms found throughout to familiarise themselves with the definitions found in the glossary section.

Contents

1 Introduction

1.1 Summary of Introduction

Section 1.2 covers an introduction to Open Face Chinese Poker, explaining the basic rules and giving illustrated examples of the game board and the scoring system. Readers unfamiliar with the game will benefit largely from reading this section in conjunction with reference to the glossary of terms.

Section 1.3 outlines the problem description, considering the limitations of existing Intelligent Agents and the reasons for these limitations.

Section 1.4 outlines specific aims and objectives for the project based on requirements for performance and meeting user demands.

Section 1.5 discusses the potential for ethical issues in this project and its area of research.

1.2 Introduction to Open Face Chinese Poker

Open Face Chinese Poker (commonly abbreviated to *OFCP*) is a perfect or imperfect information (depending on variant)[1] card game with elements of randomness due to the shuffled cards dealt. OFCP is a variant of *Chinese Poker* where players take turns placing cards face-up into three distinct *rows* (*bottom*, *middle* and *top*), creating the best possible *poker hands* they can in order to score points from each other. Stronger hands score *royalties* for extra points and royalty-scoring hands in higher rows score more points than an equivalent hand in a lower row. For example, a *Royal House* gives +25 bonus points in bottom row, or +50 in the middle row. Players score +1 point for each row they win in addition to any royalties. In the case that a player wins all 3 rows they score a *scoop* bonus which grants an additional point for each row, for a total of +6 base points before royalties are calculated. OFCP is a zero-sum game, meaning any gains by one player are balanced with losses by another player; as players win points directly from their opponents, if a player's score is +16 points then in a 1v1 game it is therefore implied that their opponent's score would be -16.

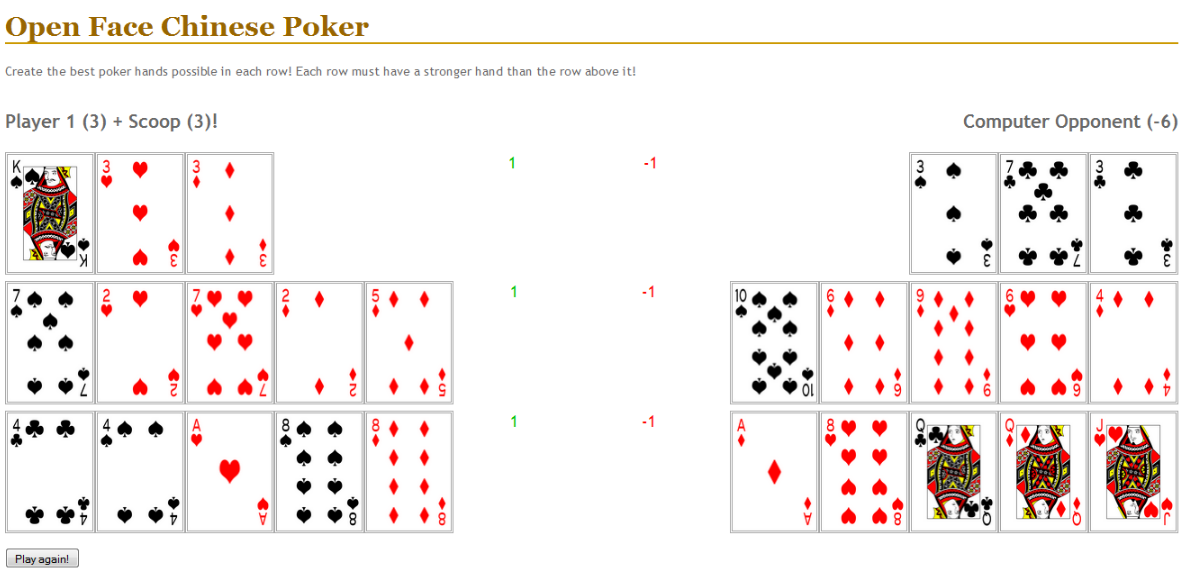


Fig 1.2.1 – example layout of the board after a single round of Open Face Chinese Poker (screenshot from an early prototype of the application)

Figure 1.2.1 shows the results and layout of the board after a game of OFCP. Player 1 wins bottom with a *Two Pair* 8s and 4s versus a *Pair* of Qs for +1 point. Player 1 also wins middle row with Two Pair 7s and

[1] Standard OFC is a perfect information game, but other variants such as *Pineapple OFC* also feature elements of hidden information

2s versus Pair of 6s for +1 point. In top row Player 1 and the computer opponent both have a Pair of 3s, and so the third card is taken into account as the *kicker*, with Player 1 clinching the win for +1 point with a King kicker versus a 7. Further to the individual row scores, because Player 1 won every row they score an additional scoop bonus of +3.

One important caveat of the game is that hands in lower rows must be stronger than those on the rows above; if a player creates a top-heavy board then that player's hand is invalid, which is known as *fouling*. When a player fouls their opponents automatically scoop them for +6 each (+1 for each row and +3 scoop bonus) in addition to any royalties, and any of the fouled player's royalties are disqualified. In the case that all players foul, no points are awarded.

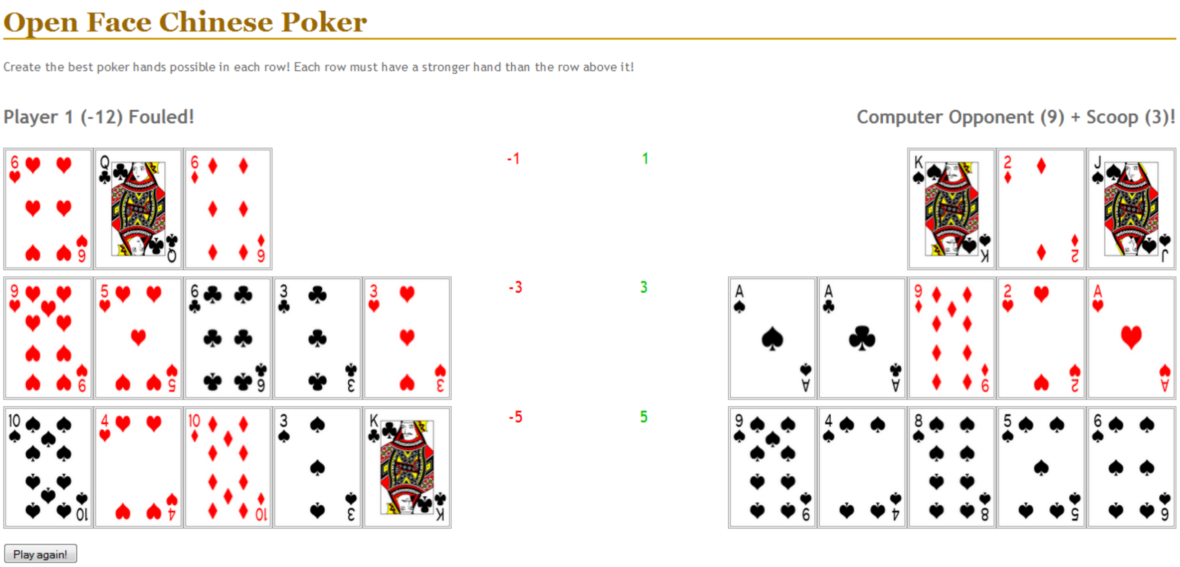


Fig 1.2.2 – Player 1 fouls and so their opponent wins an automatic scoop bonus plus royalties

In the game shown in Fig 1.2.2 Player 1 has Pair of 10s in bottom, Pair of 3s in middle and Pair of 6s in top. Because the top row contains a stronger poker hand than the row below it, the hand is invalid and the player fouls. On top of the +3 scoop bonus and points for individual row wins, the computer opponent wins further points for royalties because of its *Flush* in bottom and *Three of A Kind* in middle.

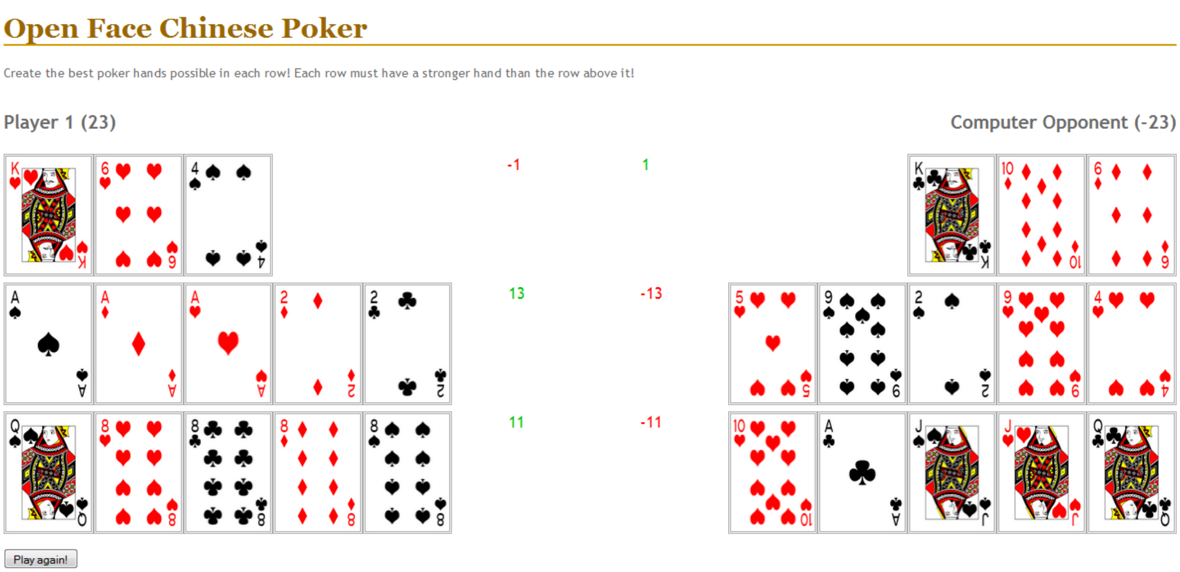


Fig 1.2.3 – Player 1 wins a lot of bonus points from royalties for strong hands

Fig 1.2.3 indicates how many points can be won from strong hands. Player 1 wins bottom row with *Four of a Kind* 8s versus the computer opponent's Pair of Jacks. Player 1 receives +1 point for winning the row, with an additional +10 points royalty for Four of a Kind on bottom. Player 1 also wins middle row with a *Full House*, Aces full of Deuces versus the computer opponent's Pair of 9s, scoring +1 point for winning the row and an additional +12 royalty. Player 1 and Player 2 both have *High Card* King in the top row, but the computer opponent wins the row because their kicker of 10 beats Player 1's 6 kicker. Player 2 wins +1 point for winning this row but the hand is not strong enough to score any additional royalties. The points each player wins are taken from their opponent, so the final score for Player 1 is $-1 + 13 + 11 = 23$, and the computer opponent's score is the inverse of this, -23.

1.3 Problem Description

Creating sophisticated Intelligent Agents for games with large branching factors poses a significant issue, as for the Agent to perform well it must be able to search many moves ahead of the current state in order to find the optimal move. With every additional level of depth searched in a game tree the complexity of the search increases exponentially due to combinatorial explosion; at a depth of d with a branching factor B there would be approximately B^d potential nodes to explore. A brute-force Depth First Search would therefore have a time complexity of $O(B^d)$ which is prohibitively complex for any reasonably exhaustive search of a game with a high branching factor. Consider for example the game of Chess which has an estimated branching factor of 35 (Mandziuk, 2010). In his influential paper *Programming a Computer for Playing Chess*, Shannon estimated a lower-bound for the game-tree complexity of Chess of 10^{120}, asserting "A machine operating at the rate of one variation per second would require over 10^{90} years to calculate the first move" (4). While in theory exhaustively analysing a game of chess from start to finish is possible, it remains implausible with any conceivable modern computer despite huge advances in processing speed and power since 1950.

Search Depth	Approximate Nodes	Approximate Time to Search (at 1 million nodes per second)
1	35	3.5×10^{-5} seconds
2	35^2	1.2×10^{-3} seconds
3	35^3	4.3×10^{-2} seconds
4	35^4	1.5 seconds
5	35^5	52.5 seconds
6	35^6	30.6 minutes
7	35^7	17.9 hours
8	35^8	26 days
9	35^9	2.5 years
10	35^{10}	87.5 years
11	35^{11}	3061 years
12	35^{12}	$\sim 100,000$ years
20	35^{20}	$\sim 2.4 \times 10^{17}$ years
30	35^{30}	$\sim 6.65 \times 10^{32}$ years
40	35^{40}	$\sim 1.8 \times 10^{48}$ years

Table 1.3.1 showing the intractability of a brute force search of a Chess game tree

Open Face Chinese Poker does not have such an astronomical space state or game-tree complexity as Chess, although unlike Chess it incorporates stochastic elements because of its nature as a card game. Considering there are $52! \approx 8 \times 10^{67}$permutations of a standard deck of cards, representation of each possible state in a game tree invokes a combinatorial explosion of branching factor. An upper-bound[2] for possible final game states is 4.96×10^{14} with an upper-bound estimate for the game tree size of 5.67×10^{33} which

[2]this estimate is generous as it includes effectively duplicate states e.g. rows with the exact same cards but in different orders

poses a similarly daunting complexity. The game tree size was calculated by taking into account each player's sequential turns for the duration of the game, using the following formula:

$$\left(\binom{8}{1}\cdot 42\right)\times\left(\binom{8}{1}\cdot 41\right)\times\left(\binom{7}{1}\cdot 40\right)\times\left(\binom{7}{1}\cdot 39\right)\times\cdots\times\left(\binom{1}{1}\cdot 28\right)\times\left(\binom{1}{1}\cdot 27\right)$$

(to be finished/ reviewed and changed)

Search Depth	Approximate Nodes	Approximate Time to Search (at 1 million nodes per second)
1	$110,208$	0.1 seconds
2	5.67×10^{33}	1.8×10^{20} years
3	5.67×10^{33}	1.8×10^{20} years
4	5.67×10^{33}	1.8×10^{20} years
5	5.67×10^{33}	1.8×10^{20} years
6	5.67×10^{33}	1.8×10^{20} years
7	5.67×10^{33}	1.8×10^{20} years
8	5.67×10^{33}	1.8×10^{20} years

Table 1.3.2 showing the possible states to consider in an exhaustive search of an Open Face Chinese Poker game tree

It is clear therefore that more processing power is not in itself a solution to the problem. While modern computers are still not powerful enough to perform such a complex and exhaustive search, competent Intelligent Agents for games like Chess have emerged regardless. The reason this is possible is because of different approaches to the problem, or through pruning algorithms which reduce the complexity of the search by removing branches of the tree that do not provide beneficial information.

Introduce problem and motivation for study, outline purpose OFCP poker bots generally underperform vs human players. Rely on simple algorithms and make sub-optimal plays. Little work done on OFCP AI, more focus on other games/ more well-known variants of poker (Texas Hold'em). Poker is an interesting area of research for game theory as has many aspects other games (e.g. chess) do not – chance from unknown cards.

Imperfect information means that more sophisticated algorithms are generally necessary for optimal play. Or can traditional algorithms with suitable heuristics perform as well or better?

Hidden information poses many challenges from an AI point of view. In many games, the number of states within an information set can be large: for example, there are 52! ~= 8x10^{67} possible orderings of a standard deck of cards, each of which may have a corresponding state in the initial information set of a card game. If states are represented in a game tree, this leads to a combinatorial explosion in branching factor.

1.4 Aims and Objectives

Specific objectives

1. Create a functional Open Face Chinese Poker Application – game environment
2. Create Intelligent Agent for application
3. AI must have a level of sophistication such that it performs well vs. humans and other AI
4. AI algorithms must be suitably optimised with tradeoffs between efficiency, space and time complexity so that it is responsive. E.g. AI calculates move in less than 5 seconds

1.5 Ethics

Ethical issues can be a concern in Poker games when playing for money; consider for example the implications of a human player unknowingly playing versus a Poker Bot. This is certainly a major concern for variants of Poker such as Texas Hold'Em which has a large online following with numerous websites and applications where virtual Poker games are played for real money. Open Face Chinese Poker on the other hand is generally only played for money in home games, with friends or at casinos as a side-game during or after a Texas Hold'Em tournament. While some websites do exist for playing OFCP for money, these do not have the same level of following as more popular variants of Poker, and the Intelligent Agent produced in this project will not provide any kind of support for integration with these websites.

In the context of the proposed application, no money will be involved and participants will be playing purely for entertainment value and research interest in Artificial Intelligence. This mitigates potential ethical issues as the project focuses not on aspects of gambling but rather on the technologies and methodologies used in creating the Web-Application and system architecture, as well as the pros and cons of various algorithms for producing a sophisticated Intelligent Agent.

2 Background

2.1 Game Theory

Game theory – John von Neumann. Traditional Game Search: e.g. 1928 neumann proposes minimax tree search. Minimax decision rules dictate that in a 2-player zero-sum perfect information game there exists strategies for each player that minimise his maximum losses (hence minimax) which must be based on considerations of all the adversary's possible responses. The strategy which minimises a player's maximum losses is called the optimal strategy.

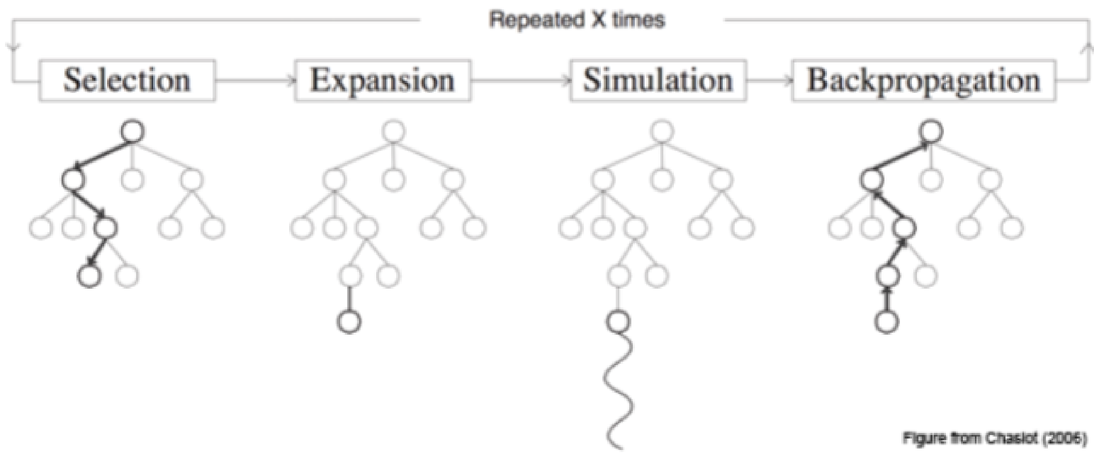
Naïve minimax vs alpha-beta pruning: AB pruning eliminates branches of the search tree where a possibility has been found which is worse than a previously examined move, meaning this branch cannot possibly influence the final decision. These traditional methods work well for e.g. chess, checkers, but are generally insufficient for more complex games that cannot be 'solved' e.g. imperfect information games such as poker.

Checkers state complexity: 10^{20} (relatively low complexity, weakly solved with traditional algorithms)
Chess state complexity: 10^{47} (higher complexity, partially solved e.g. with retrograde algorithms. May be impossible to solve with current technology)
Go state complexity: 10^{171} (very high complexity, unlikely for strong AI to come out for many years)
1940s: Monte Carlo methods formalised. 2006: Remi Coulom proposed Monte Carlo Tree Search – run random simulations and build search trees from the results.

MCTS has quickly gained traction as a strong general purpose algorithm for AI in games due to its effective results with (if properly implemented) low space and time complexity. MCTS concentrates on analysing the most promising moves, basing the expansion of the search tree on random sampling of the search space.

The game tree in MCTS grows asymmetrically, concentrating on searching more promising branches. Because of this it achieves better results than classical algorithms in games with a high branching factor. One of the most enticing benefits of MCTS is that it requires no strategic or tactical knowledge about a problem domain other than end conditions and legal moves, making MCTS implementations flexible and applicable to a variety of problem domains with little modification.

"The basic MCTS algorithm is simplicity itself: a search tree is built, node by node, according to the outcomes of simulated playouts. The process can be broken down into the following steps:



- http://www.cameronius.com/research/mcts/about/index.html "

Basic algorithm can be weak, but there are many enhancements e.g. Upper Confidence Bounds for Trees (UCT), used in 90% of MCTS applications. UCB formula:

$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$

"where vi is the estimated value of the node, ni is the number of the times the node has been visited and N is the total number of times that its parent has been visited. C is a tunable bias parameter. Exploitation vs Exploration The UCB formula provides a good balance between the exploitation of known rewards and the exploration of relatively unvisited nodes to encourage their exercise. Reward estimates are based on random simulations, so nodes must be visited a number of times before these estimates become reliable; MCTS estimates will typically be unreliable at the start of a search but converge to reliable estimates given sufficient time and perfect estimates given infinite time."

Improvements to MCTS? Light playouts – random moves. Heavy playouts utilise heuristics to influence choice of moves. "MCTS and UCT Kocsis and Szepervari (2006) first formalised a complete MCTS algorithm using UCB and dubbed it the Upper Confidence Bounds for Trees (UCT) method. This is the algorithm used in the vast majority of current MCTS implementations. UCT may be described as a special case of MCTS, that is: UCT = MCTS + UCB"

"Previous work has adapted MCTS to games which, like Spades, involve hidden information. This has led to the development of the Information Set Monte Carlo Tree Search (ISMCTS) family of algorithms (Cowling, Powley, and Whitehouse 2012). ISMCTS achieves a higher win rate than a knowledge-based AI developed by AI Factory for the Chinese card game Dou Di Zhu, and also performs well in other domains. ISMCTS uses determinizations, randomisations of the current game state which correspond to guessing hidden information. Each determinization is a game state that could conceivably be the actual current state, given the AI player's observations so far."

- http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/viewFile/7369/7595

ISMTCS useful for games like traditional Texas Hold'Em poker where each player is privy to information that others are not – i.e. their own cards. ISMTCS can model various possible game states/ permutations of what other players could have – guessing other players cards based on previous information. Not needed for OFCP because all players have the same information – cards are placed face up. However, could come into play for custom variants of OFC, such as pineapple – e.g. to model which cards are unlikely to have appeared based on how they play; if a player has a king on bottom row but does not pair it in the next hand then it is almost certain that the discarded card is not a King. Information can be built in this way to influence determined probabilities of certain cards appearing and AI can act appropriately.

2.2 Artificial Intelligence in Poker

Pot limit poker solved. Not true for holdem. Where does OFC stand? Relatively low complexity for standard OFC, but other variants of OFC are much more complex e.g. Pineapple OFC.

That said, permutations for OFCP game state: deck of 52 cards, each player places 13 cards (26 total for a heads up game)

52 choose 26 = 495,918,532,948,104 (4.9591853e+14)

Methods e,g, database look ups impractical to implement due to space complexity of game. Need a method which has a suitable compromise between time and space complexity. Monte Carlo methods are perfect for this, especially considering the usage of heuristics which can optimise the algorithm e.g. UCT, pruning tree branches

http://scrambledeggsontoast.github.io/2014/06/26/artificial-intelligence-ofcp/ - Haskell AI for OFC 'Kachushi'. Carries out monte carlo simulations of rest of game to inform expected value for decisions.

2.3 Hand Evaluation Algorithms

Naïve – non-optimal and non-trivial to implement. Simple histogram algorithm can be used to rank high card, pair, two pair, trips, full house, quads, but needs extra steps to check flushes, straights, kickers etc. -> use this approach for bespoke 3 card evaluator for top row in OFC. Very low complexity, efficiency not as much a concern as with 5 card hands.

A faster, more efficient algorithm means more hands can be evaluated more quickly leading to higher responsiveness and more optimal play – e.g. able to evaluate more hands in deeper, broader searches. Using "kmanley"'s 5 card poker hand evaluator (handles all hands properly e.g with kickers etc, reasonable efficiency compared to other algorithms, understandable, performs better than other simple naive algorithms). Written in python so can be easily used in conjunction with my backend (cherrypy server)

More efficient poker hand evaluators are available, however generally written in much more efficient languages such as C - cannot be used in conjunction with pypy.

3 Design and Implementation

use case diagrams, system architecture diagrams, wireframes

3.1 Approach

html css and javascript for website/ app. Javascript handles front end, sends POST 'reqwest's with game state to cherrypy server – backend – which validates the sent game state and then calls helper functions (python scripts) for hand evaluation, handling AI simulations etc. , updates the game state in the database, and returns appropriate response e.g. cards to be placed, scores at end of game

data flow diagram of a game of chinese poker

system flow diagram for application / system architecture diagram

The design principle adhered to throughout the process of creating the application was Rapid Application Development using Evolutionary Prototyping, creating a functional prototype which was refined and updated to meet changing requirements and to implement new features. This approach was perfect for the needs of the project as it allowed for lots of flexibility and meant that emphasis could be put on development, creating a functional or semi-functional application at each stage which implemented some of the planned features, meeting some of the requirements and being ready to build upon and develop further into a new version which improves upon itself. This flexible style is naturally advantageous over a more traditional approach such as the Waterfall model which involves rigorously defining specifications from the start, which means making changes down the line becomes increasingly difficult and costly; such a style of development was appropriated from other industries before more suitable methodologies of software development were formalised.

1. OFCP application must implement appropriate rules. E.g. correct scoring system
2. AI must have a level of sophistication such that it performs well vs humans and other AI
3. AI algorithms must be suitably optimised

4. Website should have minimal downtime

5. Website should be responsive and quick to load

6. Footprint of application must be low – if it were to be scaled up e.g. hundreds of concurrent users, server has to be able to handle this

3.2 Technologies

Website and application created with HTML, CSS and Javascript with pages created dynamically with jinja2 templates.

Python backend handling dealing of cards, processing game state, AI's card placements and scoring of game board

Pure python networking with cherrypy, which is efficient and can handle up to 1000 concurrent requests which is more than enough for the requirements of this project.

Site hosted using DigitialOcean VPS running Ubuntu 14.04 with nginx reverse-proxy.[**?**]

Game states stored in database using mongodb (which is a document-oriented database as opposed to a traditional relational database which decreases development time and reduces complexity as there is no need to constantly transform the data when reading from the database into the python backend. Mongodb is scalable and high performance, and allows for flexible data structures for example with optional values being handled trivially, with the databases getting type information from the data itself meaning they can map easily into program objects, which is specifically advantageous in this application because of the use of dictionaries to store the game state. The flexibility and ease of deployment of such a style of database makes their use well suited for web-applications such as this one) run in docker virtual environment.

Version control with github: OFCP-AI private repository. Use of version control is paramount as it allows for undesirable changes to be rolled back easily, and if something goes wrong there is always working versions available to roll back to. Using multiple branches (master and experimental) meant that a stable version could be maintained while new features were implemented safely on the experimental branch without affecting or potentially breaking the master version. Also allows for 'releases' for different versions of the software e.g. legacy client-side only, current with feature-full self-contained application

3.3 Implementation

http://www.alastairkerr.co.uk/ofc

4 Evaluation

usability tests - table of what tests, what happened

questionnaire and results, general evaluation

4.1 Unit Tests

continuous integration/ deploymentwith continuous integration when you make a commit to github it automatically connects to server and runs the tests, if they pass it deploys it http://code.tutsplus.com/tutorials/setting-up-continuous-integration-continuous-deployment-with-jenkins–cms-21511

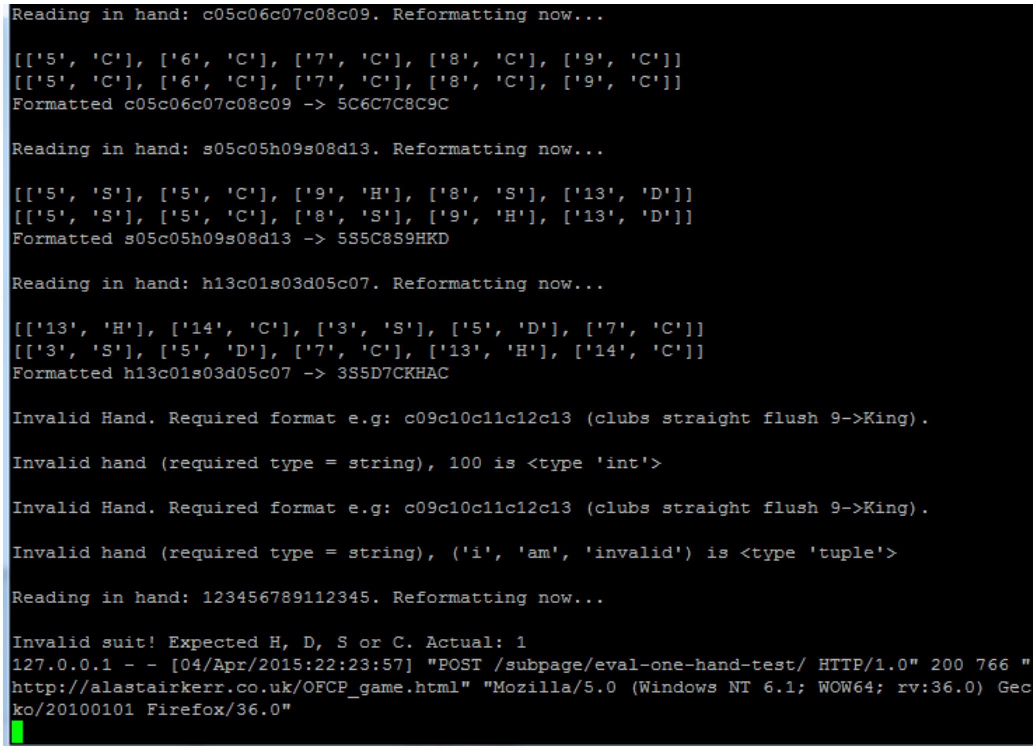
Modular testing of code e.g. individual functions using unittest python framework

```python
if (type(hand) is not str):
    print "Invalid hand (required type = string), " + str(hand) + " is " + str(type(hand)) + "\n"
    return None

if (len(hand) != 15):
    print "Invalid Hand. Required format e.g: c09c10c11c12c13 (clubs straight flush 9->King).\n"
    return None

try:
    print ('Reading in hand: ' + str(hand) + '. Reformatting now...\n' )
    cards_list = []
    formatted_hand = ""

    rank_dic = {'10':'T', '11':'J', '12':'Q', '13':'K', '14':'A'}

    for i in xrange(0,15,3):             # decode string to get each card name. index 0 -> 14 step 3
        suit = hand[i]
        rank_p1 = hand[i+1]
        rank_p2 = hand[i+2]

        suit = suit.upper()              # evaluator needs suit as uppercase char

        if suit not in ('H','D','S','C'):
            print "Invalid suit! Expected H, D, S or C. Actual:", suit
            return None

        rank = int(rank_p1 + rank_p2)    # get numerical value for rank
        if ( rank < 1 or rank > 14):
            print "Invalid rank. Accepted range 1-14.\n"
            return None
```

Fig 4.1.1 - Sample code from function 'reformat_hand_xyy_yx' in helpers.py script: use of input validation and try except blocks to catch errors

```python
test_items = ( 'c05c06c07c08c09','s05c05h09s08d13','h13c01s03d05c07','invalid',100,'fakestring',('i','am','invalid'),'123456789112345' )
for item in test_items:
    format_resp = helpers.reformat_hand_xyy_yx(item)
    if format_resp != None:
        print 'Formatted ' + str(item) + ' -> ' + str(format_resp) + '\n'
```

Fig 4.1.2 - Test inputs to ensure function works as intended



```
Reading in hand: c05c06c07c08c09. Reformatting now...

[['5', 'C'], ['6', 'C'], ['7', 'C'], ['8', 'C'], ['9', 'C']]
[['5', 'C'], ['6', 'C'], ['7', 'C'], ['8', 'C'], ['9', 'C']]
Formatted c05c06c07c08c09 -> 5C6C7C8C9C

Reading in hand: s05c05h09s08d13. Reformatting now...

[['5', 'S'], ['5', 'C'], ['9', 'H'], ['8', 'S'], ['13', 'D']]
[['5', 'S'], ['5', 'C'], ['8', 'S'], ['9', 'H'], ['13', 'D']]
Formatted s05c05h09s08d13 -> 5S5C8S9HKD

Reading in hand: h13c01s03d05c07. Reformatting now...

[['13', 'H'], ['14', 'C'], ['3', 'S'], ['5', 'D'], ['7', 'C']]
[['3', 'S'], ['5', 'D'], ['7', 'C'], ['13', 'H'], ['14', 'C']]
Formatted h13c01s03d05c07 -> 3S5D7CKHAC

Invalid Hand. Required format e.g: c09c10c11c12c13 (clubs straight flush 9->King).

Invalid hand (required type = string), 100 is <type 'int'>

Invalid Hand. Required format e.g: c09c10c11c12c13 (clubs straight flush 9->King).

Invalid hand (required type = string), ('i', 'am', 'invalid') is <type 'tuple'>

Reading in hand: 123456789112345. Reformatting now...

Invalid suit! Expected H, D, S or C. Actual: 1
127.0.0.1 - - [04/Apr/2015:22:23:57] "POST /subpage/eval-one-hand-test/ HTTP/1.0" 200 766 "
http://alastairkerr.co.uk/OFCP_game.html" "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:36.0) Gec
ko/20100101 Firefox/36.0"
```

Fig 4.1.3 - Output – invalid hands are handled properly, throw exceptions/ print usage messages rather than throwing errors

4.2 Performance of AI versus human players

Alpha testing: playing individual games with participants vs AI Playing vs experienced players, new players – get an indication of AI's comparative skill level

4.3 Performance of AI versus other AI

Pit this intelligent agent vs other AI and/or previous/ alternative versions of itself. E.g. performance of AI with MCTS vs AI using AB pruning/ minimax, MCTS vs totally random placement: if AI is working well should vastly outperform a naive AI. Visualisations of performance e.g. graphs, tables of win rates Database storing moves -> this would allow for analysis of individual rounds, games etc.

5 Conclusion

5.1 Aims and Objectives

To what extent were the aims met? Sophistication and performance of AI? Were all features implemented?

Performance of website: http://tools.pingdom.com/fpt/#!/cXxmxY/http://alastairkerr.co.uk/ofc/play/5540d2e7b878ce0

5.2 Reflection

Reflection on project, decisions, performance etc.

Design - frontend is functional but could have been designed better from the start. e.g. originally hard-coding player's card image objects rather than dynamically creating them with javascript

Backend works well but if different technologies and languages were used could be more efficient - e.g. hand evaluators written in C using bitwise operators could evaluate millions of hands per second rather than hundreds of thousands - could shave off seconds of processing time which could either lead to increased responsiveness or allow for more games to be simulated by the AI making it more likely to find optimal solutions for hand placements.

The choice of a flexible software methodology worked well overall because of the evolving requirements and design choices, as well as the individual nature of the project. In comparison, in a large team of developers issues with this approach could arise from lacking a clear design focus and having limited control - a necessary trade-off that is an inevitable consequence of the increased flexibility this methodology enables. One important pitfall to avoid with Rapid Application Development is focusing too much on individual components without getting a clear view of the system's design, making minor changes without considering possibilities for an improved design structure. While there is generally a clear design phase before entering the initial implementation stage, there can be a tendency to omit a renewed design phase in subsequent implementation cycles leading to a lack of documentation which can have large consequences down the line; as Gerber et al. (2007) state in the analysis of one case study "... due to the fact that the design was not formally documented and reviewed, the discrepancy was only discovered after the implementation phase. This situation caused conflict between developers and analysts and in the end necessitated a redesign effort which put unnecessary pressure due to time constraints and limited resources on the whole development team". Design choices in early prototypes had a carry-on effect which meant that later down the line code refactoring was necessary in order to create a more coherent system structure, which potentially could have been avoided or reduced with a stricter design philosophy.

An apt example of this is seen in legacy prototypes of the application, which were client-side only. This was a choice that was made in order to quickly create a functional prototype, using JavaScript to simulate processes that would be handled elsewhere in the final application's architecture (such as dealing cards). This was useful because it resulted in a functional application which implemented some of the planned features, leading to a clearer understanding of the needs of the project, but had to be adapted later in order to create a more logical system structure which could meet the requirements, such as backend processing to handle the game states and calculate the Intelligent Agent's moves.

5.3 Improvements

add a new game button!

What can be done to improve the application/ AI in the future?

Limitations: "With any method based on random simulation, it is inevitable that poor quality moves will be chosen with nonzero probability, due to a particularly lucky run of simulations making the move appear better than it is. " - http://www.aaai.org/ocs/index.php/AIIDE/AIIDE13/paper/viewFile/7369/7595 page 5

Due to the need for a compromise between finding the optimal solution and finishing the request in a reasonable amount of time the number of simulated games is limited and therefore it is possible that sub-optimal plays will be over-valued due to the element of randomness.

Improve frontend – make the app more visually appealing

Multiplayer support for players vs players as well as players vs AI(s) or players vs players vs AI(s)

Add support for other variants of OFC such as pineapple - includes elements of hidden information as players choose to discard one card, the identity of which is known only to them. Will increase complexity of game requiring further modifications/ improvements to Agent, and would make any performance issues even more of an obstacle.

The Intelligent Agent generally favours optimal card placements, but because of the element of randomness in evaluating moves there is the potential for the AI to mistakenly over-value a sub-optimal move. As Whitehouse et al. (2013) state "With any method based on random simulation, it is inevitable that poor quality moves will be chosen with nonzero probability, due to a particularly lucky run of simulations making the move appear better than it is". This potential for inaccurate evaluation of a move's strength has an inverse relationship with the number of iterations of simulated games - as the number of iterations increase the result diverges to the optimal solution, meaning that with an infinite amount of simulations the probability of finding the best move is 1. The performance of the Intelligent Agent therefore could be improved with more processing power and/or a longer allocated time to simulate games, although it is important to note that there are diminishing returns with this strategy; doubling the iterations does not mean that the results will be twice as good.

This ties in with the choices made for the implementation and configuration of the Agent, specifically in regards to compromises between responsiveness and finding the best move - in the specified requirements the Agent was intended to take no more than 5 seconds to calculate its turn, and the application satisfies this requirement. However, increasing the iterations would make satisfying this criteria infeasible without increased processing power or through further optimisation. This could be achieved in various ways, for example by rewriting the application in a more efficient language such as C, or by implementing more advanced heuristics to reduce the complexity of the calculations, or using a different algorithm such as a more advanced implementation of the Monte Carlo method like Monte Carlo Tree Search with UCT, as discussed in Section 2 of this dissertation. Overall the implementation of the Intelligent Agent satisfies the specified requirements and works well for its intended purpose in the scope of this application, but for larger scale implementations would likely need to make use of one or more of these changes in order to achieve increased scalability, for example for use in a commercial application with thousands of concurrent users.

6 Glossary of Terms

6.1 Poker Variants

- **Texas Hold'Em** is a popular variant of Poker where each player receives 2 cards for use individually in combination with 5 community cards shared between all players, with players combining any of their available cards in order to create the strongest standard poker hand possible
- **Chinese Poker** is a variant of poker where players are dealt 13 cards which they must arrange into three rows, placed face down. Players announce in clockwise order whether or not they wish to play their hand, and then all players announce their royalties and show their cards
- **Open Face Chinese Poker** is a variant of Chinese Poker where players act in clockwise order, receiving first 5 cards which are placed face up and then one card at a time until all players have placed 13 cards. Players must create valid hands consisting of stronger poker hands in lower rows, and score points from their opponents for winning corresponding rows. Additional points known as royalties can be won for particularly strong hands
- **Pineapple OFC** is a sub-variant of Open Face Chinese Poker, following the same basic rules as the standard variant with the distinction that in subsequent rounds after the initial 5 card placements, players receive 3 cards and choose to place 2 and discard 1. This introduces higher action play as well as elements of hidden information as other players are unaware of which cards their opponents have discarded, although there is potential to infer this information based off of how the player acts

6.2 Poker Hands Guide (Weakest to Strongest)

- **High Card** is the lowest ranking poker hand and is the default when no other hands have been made. An example High Card is the Jack of Spades, which would beat any hand comprised of a High Card ranked 10 or less, but would lose to a High Card Queen, King, Ace, or any stronger poker hand.
- **Pair** is the second weakest poker hand. A player has a pair when they have two cards of the same rank, for example the 7 of Hearts and 7 and Clubs would form a Pair of 7s.
- **Two Pair** is the next strongest hand rank, consisting of two different pairs. For example having the 5 of Spades, 5 of Diamonds, 9 of Hearts and 9 of Clubs would form Two Pair 9s and 5s. When comparing a Two Pair to another Two Pair the highest ranked pair takes precedence.
- **Three of a Kind** or a **Set** beats Two Pairs, Pairs and High Cards and consists of three same-ranked cards, such as the Ace of Spades, Ace of Diamonds and Ace of Hearts, which would form Three of a Kind Aces

- **Straight** is a hand where a player has 5 sequential cards, such as 4 of Diamonds, 5 of Hearts, 6 of Hearts, 7 of Clubs, 8 of Spades, which would form a Straight 8 High. The lowest ranked straight spans Ace to 5 and the highest ranked straight spans 10 to Ace. It is important to note that straights do not wrap around; you cannot form a straight such as Queen, King, Ace, Deuce, 3.
- **Flush** is one of the stronger poker hands, consisting of cards which are all the same suit. For example the 6 of Hearts, 9 of Hearts, Jack of Hearts, Queen of Hearts and King of Hearts would form a King High Flush.
- **Full House**, sometimes known as a **Boat**, is one of the strongest poker hands available, and comprises both a Three of a Kind and an additional Pair. For example Three of a Kind Tens with Pair of Jacks would combine to form a Full House, Tens full of Jacks.
- **Four of a Kind** is a hand obtained when a player has every instance of a particular card rank, such as King of Hearts, King of Diamonds, King of Clubs and King of Spades which would form Four of a Kind Kings.
- **Straight Flush** is effectively the strongest poker hand possible, consisting of 5 sequential same suited cards. For example the 4 of Spades, 5 of Spades, 6 of Spades, 7 of Spades and 8 of Spades would form a Straight Flush 8 High.
- **Royal Flush** is a special instance of a Straight Flush where a player has the 10, Jack, Queen, King and Ace of a particular suit. Royal Flushes are particularly rare; in Texas Hold'Em the probability of getting a Royal Flush is approximately 0.000154%.

6.3 Open Face Chinese Poker Terminology

- A **row** is a set structure for placing cards. At the end of a game when scoring occurs poker hands in each player's rows are compared to the opposing player's hand in their corresponding row. There are three different rows as described below
- **Bottom Row** or **Back Hand** is the foundation row, and consists of 5 cards. Out of all three rows this must have the strongest poker hand or the player fouls.
- **Middle Row** or **Middle Hand** also consists of 5 cards. It must have a weaker hand than Bottom Row in order for the player's hand to be valid.
- **Top Row** or **Front Hand** consists of 3 cards, meaning the best possible hand here is a Three of a Kind (in most variant 3 card straights and flushes do not count). Top Row must have a weaker hand than both middle and bottom row.
- **Scoop** is a bonus awarded to a player when they win all 3 rows against an opponent. On top of the standard +1 point per row won, the player is granted an additional bonus of +3 points which is also taken from their opponent.
- **Fouling** occurs when a player plays an invalid hand, for example by putting a stronger hand in their middle row than their bottom row. When a player fouls any of their royalties are null, and their opponent is automatically awarded a scoop bonus so long as their hand is valid.
- **Kicker** is the term used to describe the next card taken into account when comparing two otherwise equal hands. For example, if both Player 1 and Player 2 have a Pair of 8s then the rest of their cards would be considered, and whichever player has the highest rank wins. If both players kicker is equivalent then the next highest kicker will come into play and so on.
- **Royalties** are bonus points awarded to player's for particularly strong hands. Just like any other points a player wins, they are taken directly from opposing players. Hands in higher rows score higher royalties than equivalent hands in lower rows. For example, a Full House in bottom row is worth 6 points in bottom or 12 points in middle. Another example is that a Three of a Kind in bottom row scores no royalty, but gets 2 bonus points in middle and between 10 and 22 points in top depending on

the rank (Three of a Kind Deuces scores 10 points up to Three of a Kind Aces with 22 points). See here for a full list of royalties: http://www.wsop.com/2013/Open_Face_Chinese_Structure_Sheet.pdf

6.4 Computer Science and Mathematical Terminology

- **Algorithms** are precisely defined step-by-step instructions describing a set of operations to be performed
- **Artificial Intelligence** is a field of study in Computer Science which focuses on simulating intelligent behaviour in computers.
- **Automated Planning and Scheduling** is a branch of artificial intelligence which focuses on strategies and actions to be performed by Intelligent Agents or autonomous robots. Research into optimisations and improved heuristics is a key area of interest due to the potentially huge complexity of problems and solutions.
- The **Branching Factor** is the average amount of branches from a node in a tree, indicating the tree's complexity. At a depth of d with a branching factor of B there would be approximately B^d nodes.
- **Combinatorial Explosion** is the phenomenon of exponential growth encountered in search problems. Deeper searches become increasingly complex increasing by the branching factor at each level
- A **Game Tree** represents possible states in a game, with each node representing a possible position and each edge representing a possible move
- **Game Theory** is a branch of mathematics focused on determining optimal strategies and decisions in competitive situations
- **Heuristics** are techniques used when classical approaches would fail or be prohibitively slow to complete. Heuristics use intelligent guesses to reduce the complexity of problems. For example in the context of a Chess game rather than trying to explore every single possible move and subsequent tree branch, a heuristic based approach would ignore branches starting with clearly bad moves
- **Hidden Information** is relevant information available to one or more agents but not others. For example in a Poker game, the cards that players individually possess or have discarded are known to them but are hidden from other players.
- An **Intelligent Agent** is an autonomous software entity that perceives and acts upon its environment, performing reasoning in order to solve problems and determine actions, exhibiting goal-oriented behaviour.
- The **Optimal Move** or **Optimal Strategy** is the move or strategy that will lead to the most favourable outcome for an Agent
- **Perfect Information Games** are games where at any stage all relevant information is available to an agent in order to inform its decision. In comparison in an **Imperfect Information Game** certain information about the game state or prior actions are unknown.
- **Rule Based Systems** store expert knowledge for a particular domain as a set of rules which can be used in various artificial intelligence applications. Rule Based Systems are used extensively for a wide range of purposes such as game playing, credit card authorisation and fraud detection.
- **Solved Games** are games where the outcome (Win/Loss/Draw) can be predicted at any stage assuming optimal play by all players
- A **Zero-Sum Game** is a game in which a player's gains are losses are balanced by another player's gains or losses. Open Face Chinese Poker is an example of a Zero-Sum Game as a player's gains are directly taken from their opponent.

7 References

- Mandziuk, J., 2010, Knowledge-Free and Learning-Based Methods in Intelligent Game Playing, p3
- Shannon, C., 1950, Programming a Computer for Playing Chess, Philosophical Magazine, Ser .7, Vol. 41, No. 314, p4
- Browne, C. Et al, 2012, A Survey of Monte Carlo Tree Search Methods, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 1.
- Cowling, P. Et al, 2012, Information Set Monte Carlo Tree Search, IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES, VOL. 4, NO. 2.
- Whitehouse, D. Et al, 2013, Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game, University of York, p5
- Özcan, E. and Hulagu, B., 2004, A Simple Intelligent Agent for Playing Abalone Game: ABLA, Proc. of the 13th Turkish Symposium on Artificial Intelligence and Neural Networks, pp. 281-290
- http://scrambledeggsontoast.github.io/2014/06/26/artificial-intelligence-ofcp/ - Intelligent Agent 'Kachushi' for Open Face Chinese Poker using Monte Carlo methods, implemented in Haskell, accessed 2015
- https://www.digitalocean.com/community/tutorials/how-to-deploy-python-wsgi-applications-using-a-cherrypy-web-server-behind-nginx
- Gerber, A. Et al, 2007, Implications of Rapid Development Methodologies, CSITEd 2007, p242-243